下载APP

10｜带宽预测：3大算法准确预估网络带宽

2021-12-13 李江

《攻克视频技术》 课程介绍 >

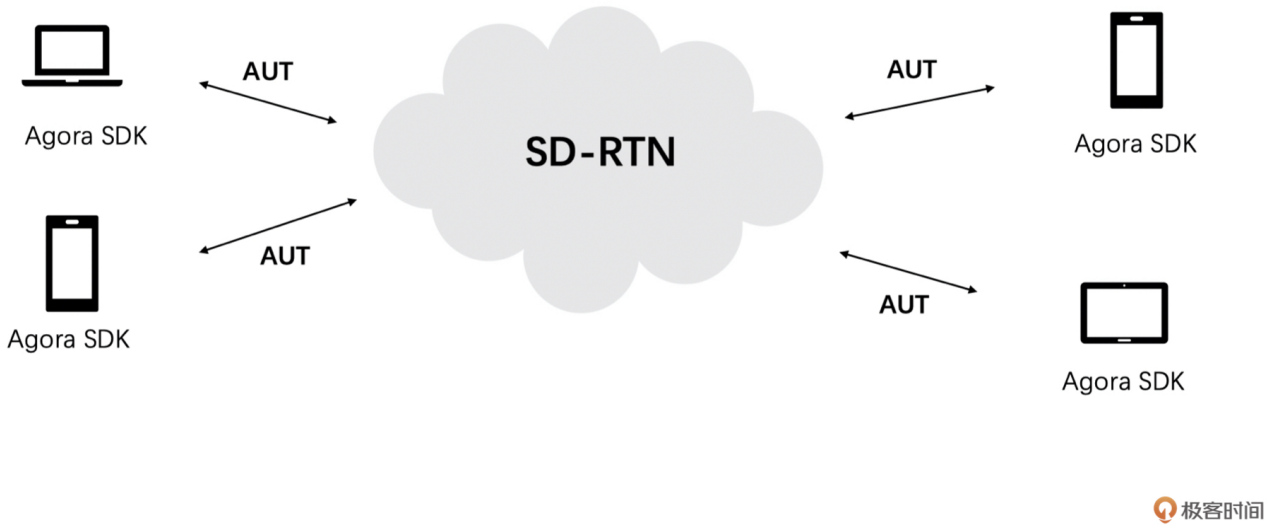
讲述：李江
时长 26:01 大小 23.84M

你好，我是李江。

上一节我们详细地讲述了 RTP 和 RTCP 协议。RTP 协议用来封装传输的音视频数据并带上一些基本的信息，而 RTCP 协议则用来统计这些 RTP 包的传输情况。RTP 和 RTCP 一般是使用 UDP 协议作为传输层协议的。因为音视频数据需要比较高的实时性，TCP 协议不太适合，所以我们一般使用 UDP 协议。但是 UDP 协议没有实现拥塞控制算法。因此，我们使用 UDP 协议作为传输层协议的话，需要自己实现拥塞控制算法。

比如说，我们声网就是自己实现了一个全球实时通信网 SD-RTN，并研发了 Agora Universal Transport（AUT）传输算法。我们的 SD-RTN 和 AUT 内部实现了适合不同网络模型的拥塞控制和丢包重传等一整套高质量的传输算法和策略。如果你使用了我们的音

视频 SDK，则无需自己关注拥塞控制和丢包重传等一系列弱网对抗算法，SD-RTN 和 AUT 会保证你在进行音视频通信时的流畅度和实时性要求。



一般情况下，音视频场景中的拥塞控制和丢包重传等算法的基础就是 RTP 和 RTCP 协议。我们需要通过 RTP 包的信息和 RTCP 包中传输的统计信息来做拥塞控制和丢包重传等操作。因此，我再强调一下，上一节课是我们之后几节课的基础，你需要完全掌握。

今天，我们主要讨论拥塞控制中的带宽预测算法，在之后的课程中我们还会介绍码控算法和丢包重传算法。带宽预测几乎是整个音视频传输和弱网对抗中最重要的环节，也是最难的一个环节。因此，你可能需要多看几遍今天的课程才能完全掌握。

而且，带宽预测算法涉及到很多阈值设置等细节问题。为了能够更好地带你了解带宽预测的思想，我们在讨论的过程中会将一部分细节省略，这样更加方便你理解核心思想。同时，由于 WebRTC 在实时音视频中占据绝对的领导地位，它的带宽预测算法非常的成熟，因此，我们今天将以 WebRTC 的带宽预测算法作为讨论的对象，对其进行深入剖析和探讨。

带宽预测

带宽预测，顾名思义，就是实时预测当前的网络带宽大小。**预测出实际的带宽之后，我们就可以控制音视频数据的发送数据量。**比如说，控制音视频数据的编码码率或者直接控制

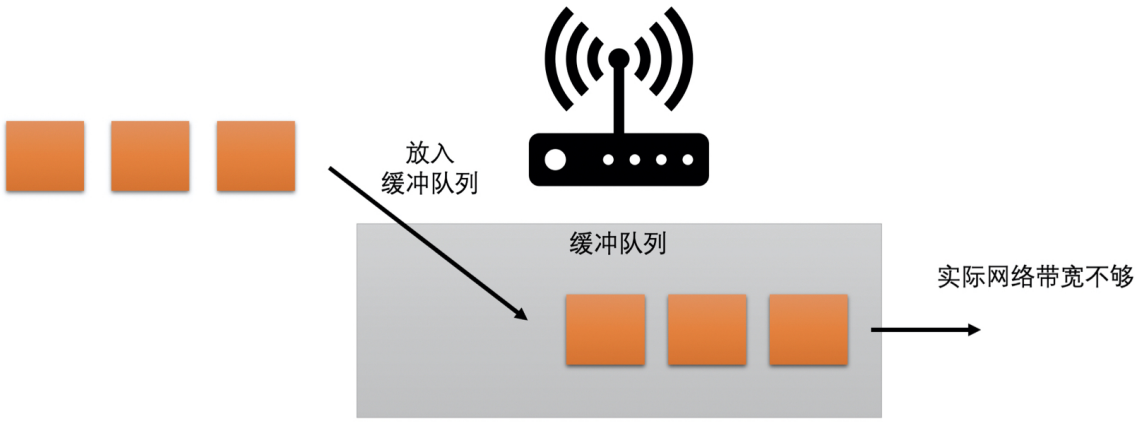
发送 RTP 包的速度，这都是可以的。控制住音视频发送的数据量是为了不会在网络带宽不够的时候，我们还发送超过网络带宽承受能力的数据量，最后导致网络出现长延时和高丢包等问题，继而引发接收端出现延时高或者卡顿的问题。因此，带宽预测是非常重要的。

而现在的网络中，大多存在两种类型的网络设备：一种是有较大缓存的；一种是没有缓存或者缓存很小的。

前者在网络中需要转发数据过多的时候，会把数据先缓存在自己的缓冲队列中，等待前面的数据发送完之后再发送当前数据。这种情况就会在网络带宽不够的时候，需要当前数据等一段时间才能发送，因此**表现出来的现象就是网络不好时，延时会加大。**

后者在网络中需要发送的数据过多的时候，会直接将超过带宽承受能力的数据丢弃掉。**这种情况就会在网络带宽不够的时候，出现高丢包的现象。**

通过下面的图你可以很容易理解：



较大缓冲区网络设备

极客时间

因为互联网中这两种类型的网络设备都存在，为了能够兼顾这两种类型的网络，WebRTC 中设计了两个主要的带宽预测算法：一个是基于延时的带宽预测算法；一个是基于丢包的带宽预测算法。下面我们逐一讨论一下。

基于延时的带宽预测算法

基于延时的带宽预测算法主要是通过计算一组 RTP 包它们的发送时长和接收时长，来判断当前延时的变化趋势，并根据当前的延时变化趋势来调整更新预测的带宽值。比如说：

> 如果延时有明显变大的趋势就说明实际带宽值应该比当前的发送码率要小，则需要降低预测的带宽值。
>
> 如果说延时没有变大，说明当前带宽良好，可能实际带宽值比当前的发送码率还要大，则可以提高预测的带宽值，直到延时有明显变大的趋势再降低预测的带宽值。我们不断这样实时调整更新带宽，来实现带宽预测。

从上面的讲述我们可以看到，基于延时的带宽预测算法，主要有 4 个步骤：

1. 计算一组 RTP 包的发送时长和接收时长，并计算延时；
2. 需要根据当前延时和历史延时的大小来计算延时变化的趋势；
3. 根据延时变化趋势判断网络状况；

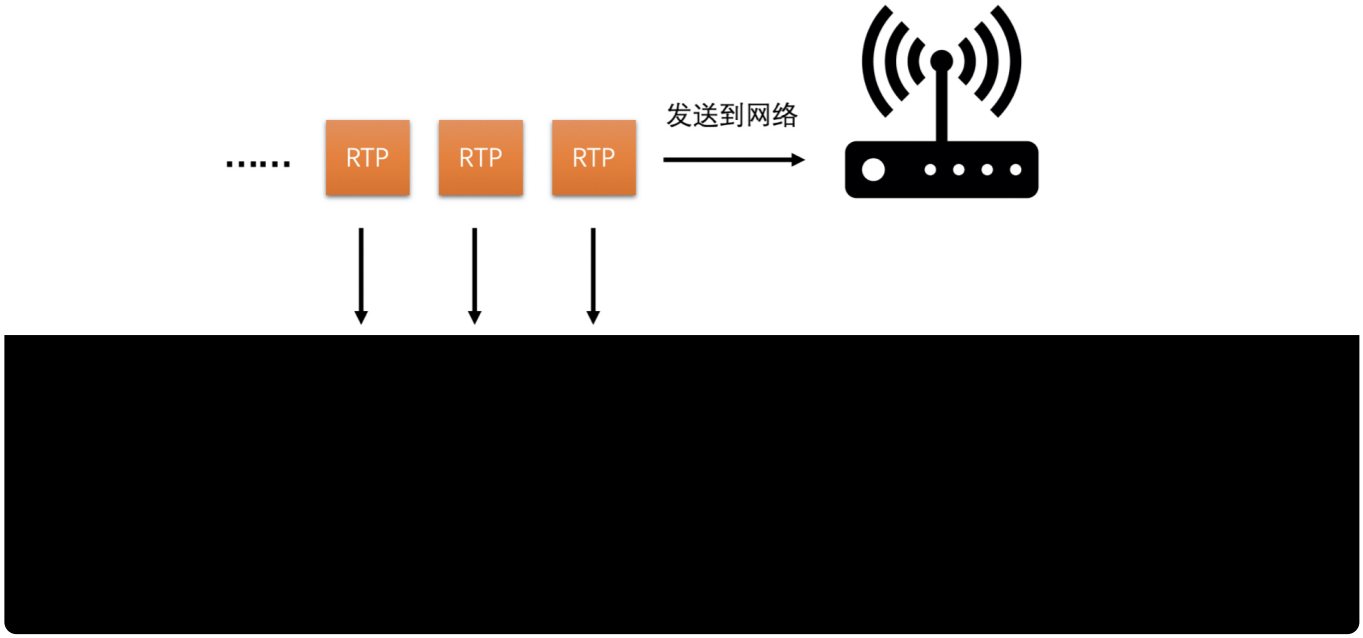
4. 根据网络状况调整更新预测带宽值。

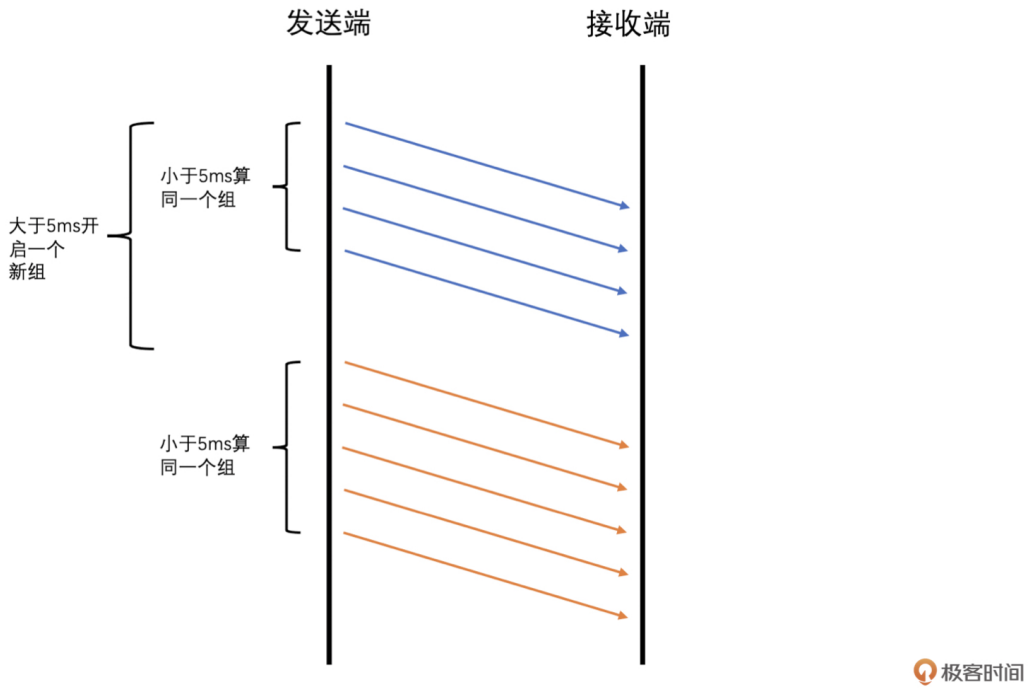
下面，我们一个个步骤来展开详细讲解。

计算延时

WebRTC 中计算延时的时候是将 RTP 包按照发送时间来分组的，并且要求当前组中的第一个包和最后一个包的发送时间相差不能大于 5ms，而大于 5ms 则是新的一组的开始。同时，由于 UDP 会出现包乱序到达的情况，可能导致后面包的发送时间比前面包的还小。为了防止这种情况的发生，我们要求乱序的包不参与计算。

并且发送端在发送每一个 RTP 包的时候会记录每一个包的包序号和实际发送时间，并把这些信息记录到一个发送历史数据里面方便之后计算使用（这个发送历史数据好几个计算需要用到，挺重要的）。





同时，接收端收到每一个包的时候也会记录包的包序号和实际的接收时间。每隔一段时间就会将

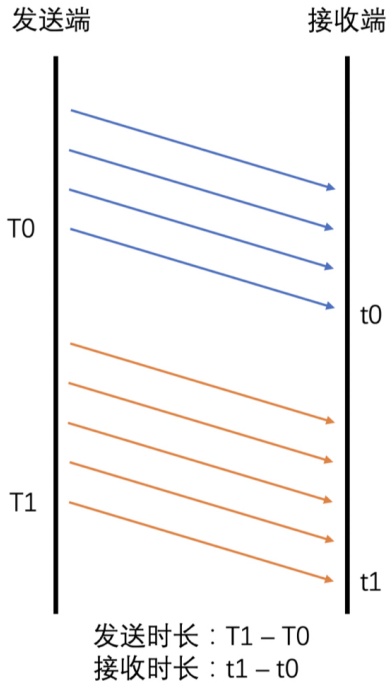
这些统计信息发送到发送端。现在的 WebRTC 版本中接收端是通过 RTCP 协议的 Transport-CC 报文反馈接收信息的，这个报文主要包含两个信息：

1. 每一个包序号对应的包是不是接收到了；
2. 实际的这个包相比前一个包的接收间隔。

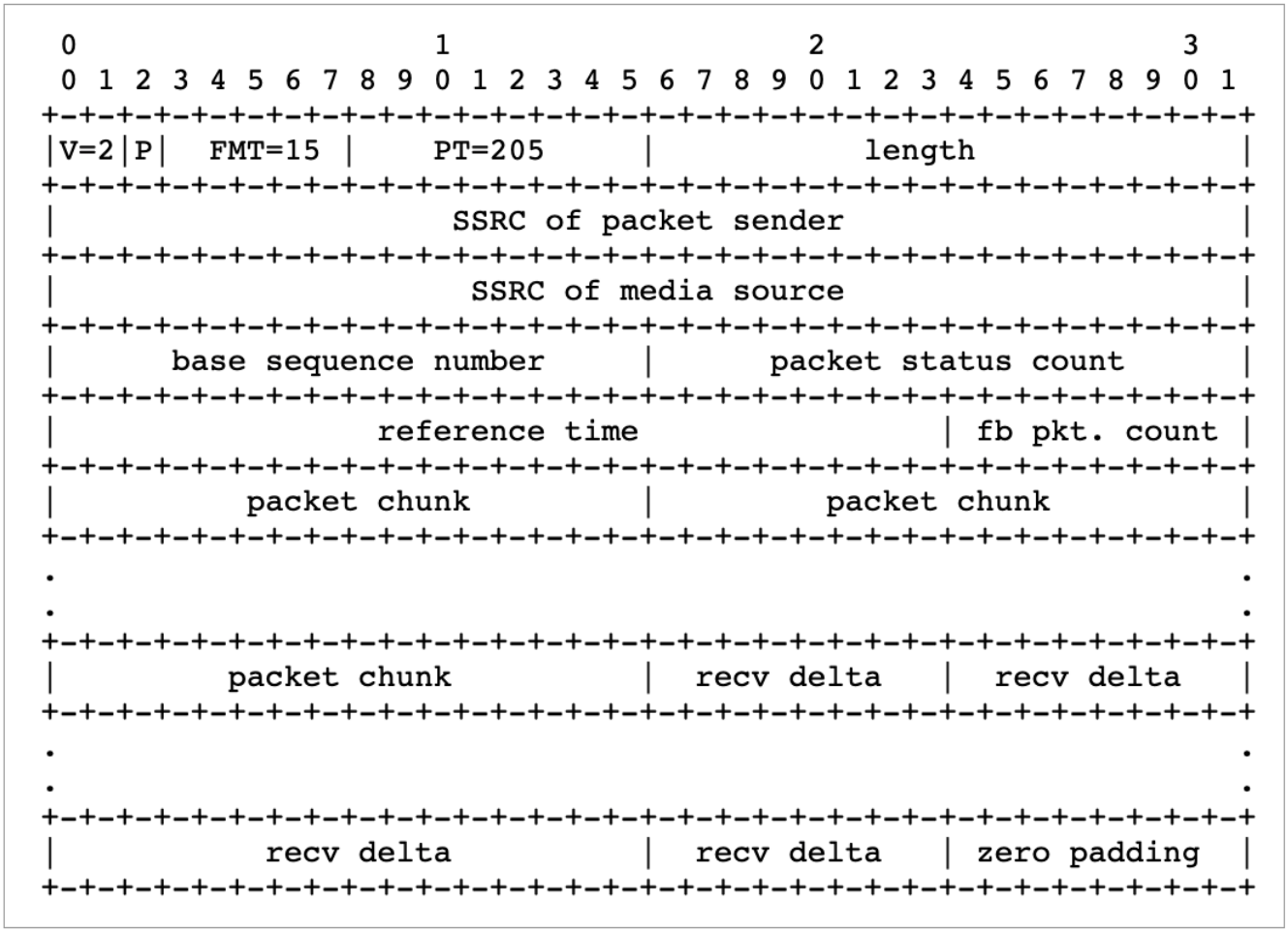
接收端将这些信息组成 Transport-CC 报文发送给发送端。

如果发送端收到这个报文，就可以知道每一个 RTP 包有没有接收到了。如果没有接收到就是丢包了。同时也可以知道没有丢失的 RTP 包的接收时间。

发送端就可以根据发送历史数据中各个包的发送时间和 Transport-CC 报文中计算得到的各个包的接收时间，来计算出前后两组包之间的发送时长和接收时长了。计算方法如下。



其中 Transport-CC 报文格式可以参考下图，具体各字段的解释可以参考这个RFC 文档和这篇文章。总的来说，就是通过 Transport-CC 报文，我们可以计算得到每一个包是不是丢失了，以及没有丢失的每一个包的接收时间。这里我们不做过多展开。



图片来源RFC文档

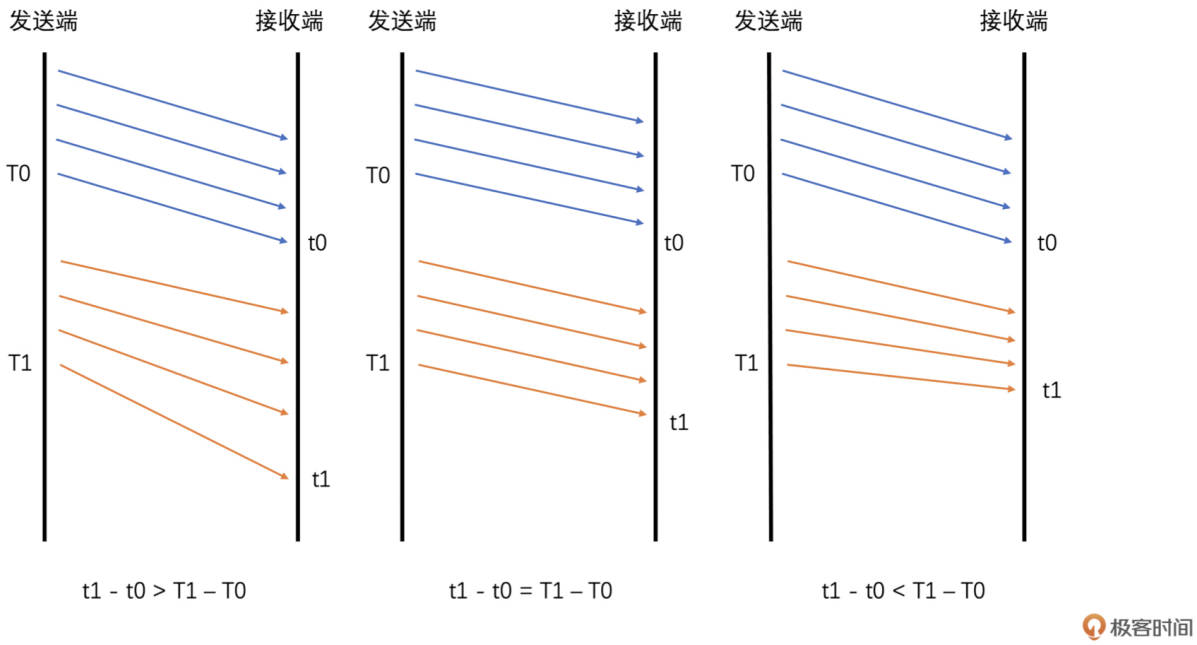
有了发送时长和接收时长，我们**将接收时长减去发送时长就是延时**了。

如果接收时长大于发送时长，延时就大于 0，说明当前网络有点承受不了当前的发送数据量，产生了缓存，继而产生了延时。

如果接收时长几乎等于发送时长，延时就几乎为 0，说明当前网络可以承受当前的发送量，因此几乎没有延时。

如果接收时长小于发送时长，也就是延时小于 0 呢？这种情况比较特殊，一般出现在之前因为网络带宽不够已经缓存了一部分数据，但是网络在明显变好，从而网络设备快速地将缓存中的数据发送出去的时候。这种情况下就会出现接收时长很短，导致接收时长还小于发送时长，这个时候延时就是一个负数。

这三种情况具体可参考下图：



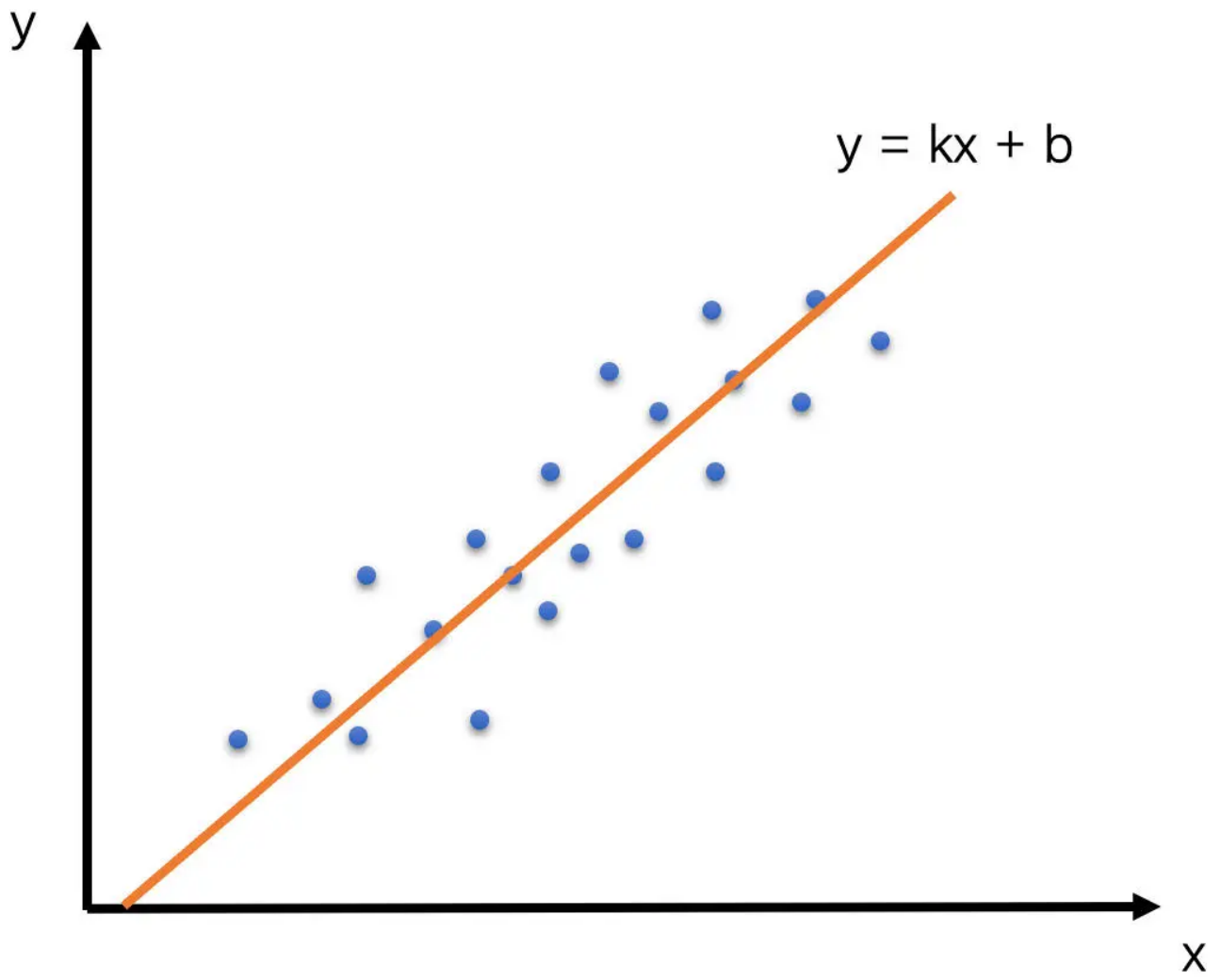
好了，这就是延时的计算方法。有了延时之后，我们还不能直接使用这个延时来判断网络的好坏，因为网络变化很快而且存在噪声，有的时候延时会因为网络噪声突然变大或变小。因此，**我们需要通过当前延时和历史延时数据来判断延时变化的趋势，来平滑掉网络噪声引起的单个延时抖动。**

说到这里就必须引入一个重要的滤波器了，那就是 Trendline Filter。这个滤波器就是用来计算得到延时变化趋势的。

延时变化的趋势计算

Trendline Filter 中保存了 20 个最近的延时数据，这些延时数据跟前面直接计算的延时还不完全一样。它们包含了两个部分：一个是当前这个 RTP 包组所属的 Transport-CC 报文到达发送端的时间；另一个是经过平滑后的累积延时，它是通过前面计算得到的延时和历史累积延时加权平均计算之后得到的。这样也可以一定程度上防止延时波动太大的问题。

在 WebRTC 看来，如果设 RTP 包组所属的 Transport-CC 报文的到达时间为 x，累积延时为 y 的话，那么 x 和 y 应该是呈线性关系，也就是说 y = kx + b。Trendline Filter 就是使用 20 个延时数据，通过线性回归的方法，求得其中的 k 值，也就是斜率。这个斜率就表示延时变化的趋势。其中线性回归的求解方式用的是最小二乘法。示意图和求解公式如下：



最小二乘法求k：

$$k = \frac{\sum_{i=0}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=0}^{n}(x_i - \bar{x})^2}$$

$\bar{x}$和$\bar{y}$表示x和y的平均值

延时趋势k的求解过程

首先求得延时:

$$delay_i = (t1 - t0) - (T1 - T0)$$

求累计延时:

$$acc_delay_i = acc_delay_{i-i} + delay_i$$

$$smooth_delay_i = 0.9 * smooth_delay_{i-i} + 0.1 * acc_delay_i$$

求$Transport - CC$达到时间平均值:

$$\bar{x} = \frac{\sum_{j=0}^{i} arrival_time_j}{i}$$

求累计延时平均值:

$$\bar{y} = \frac{\sum_{j=0}^{i} smooth_delay_j}{i}$$

求延时趋势k:

$$k = \frac{\sum_{j=0}^{i}(arrival_time_j - \bar{x})(smooth_delay_j - \bar{y})}{\sum_{j=0}^{i}(arrival_time_j - \bar{x})^2}$$

当斜率 k > 0 时，表示有延时；当 k = 0 时，表示几乎没有延时；当 k < 0 时，表示不仅没有延时，反而接收速度更快。

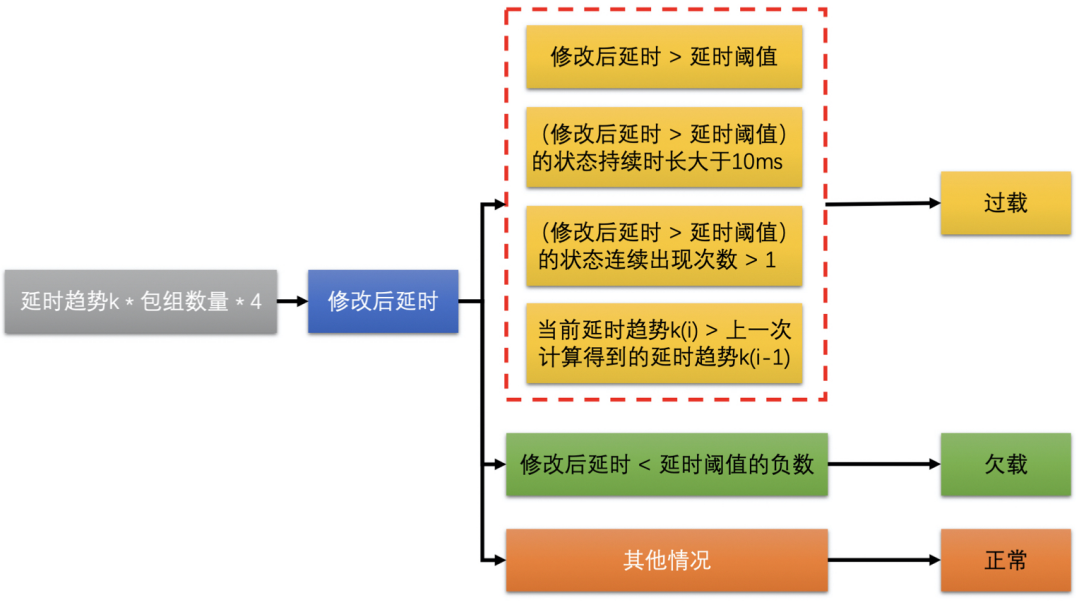
当然了，我们计算出来的延时变化趋势还不能直接说明网络当前的变化方向。因为网络是变化无常的，不能因为测到延时稍有变大就认为网络变差，延时稍有变小就认为网络变好。我们需要一个根据当前延时趋势和延时阈值，来判断网络是不是真的变好和变坏的模块，这个模块就是我们下一个讨论的重点，叫做过载检测器。

网络状态判断

过载检测器有两个主要的工作：一个是通过当前的延时趋势和延时阈值来判断当前网络是处于过载、欠载还是正常状态；一个是通过当前的延时趋势来更新延时阈值，是的，延时阈值不是静态不变的，阈值是跟着延时趋势不断自适应调整的。

我们下面先来看看网络状态的判断，之后再来讨论一下阈值的更新。

网络状态的判断其实比较简单。就是将延时趋势 k 乘以一个固定增益 4 和包组的数量（包组数量最大是 60）作为当前的修改后延时值。将当前的修改后延时值跟延时阈值进行比较，然后根据比较的结果来判断网络状态。步骤如下图所示：



得到网络状态之后，我们会使用当前的修改后的延时值去更新当前的延时阈值。为什么需要这样做呢？

我认为，延时阈值的更新是因为网络是不断变化的，延时变化也很快，而有的时候延时很大，有的时候又很小。为了防止阈值太大，网络状况检测不够灵敏，同时也防止阈值太小，网络状况检测太敏感了。所以**延时阈值会随着当前的延时做缓慢的调整**。其调整的公式如下：

延时阈值$threshold$的更新过程:

$$threshold_i = threshold_{i-1} + \Delta T * alpha * (|修改后延时| - threshold_{i-1})$$

$$alpha = \begin{cases} 0.0087 & |修改后延时| < threshold_{i-1} \\ 0.039 & 其他情况 \end{cases}$$

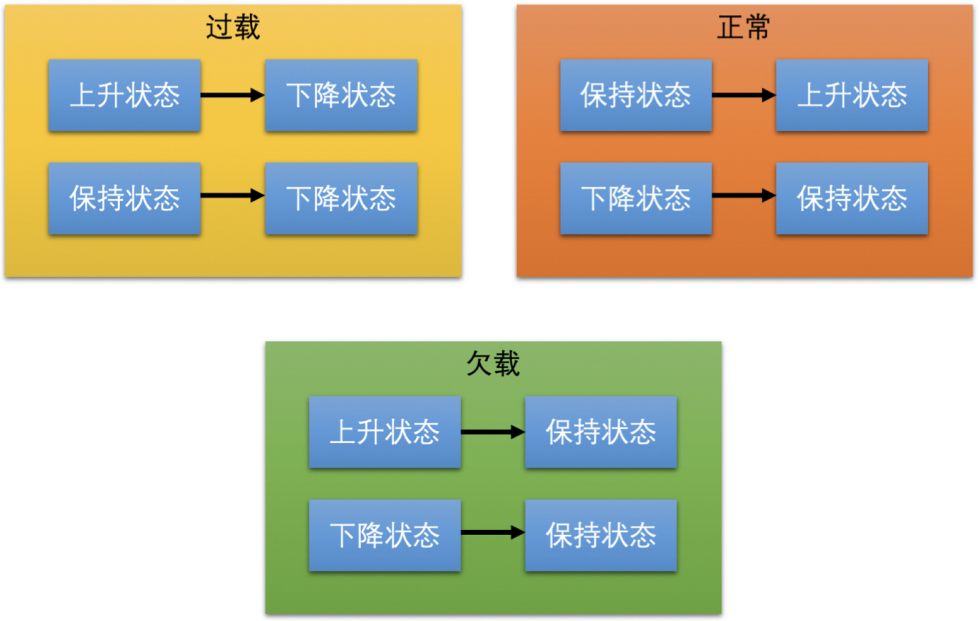
ΔT表示本次更新距离上一次更新的时间差

有了修改后延时和延时阈值之后，我们就可以计算网络状态了。又根据网络所处的具体状态，我们就可以调整更新当前的带宽值了。而负责更新带宽值的模块就是速率控制器。下面我们就来详细讨论一下如何将网络状态转化到实际带宽的调整上去的吧。

带宽调整更新

速率控制器的主要工作就是更新预测带宽值。它里面维护着一个状态机。这个状态机主要是用来干什么的呢？其实，**状态机主要用来根据过载检测器检测到的网络状态和状态机目前所处的状态，来更新状态机的状态的。**

状态机有三个状态，分别是上升、保持和下降状态。当处于上升状态时，速率控制器需要提升带宽值；当处于下降状态时，需要降低带宽值；当处于保持状态时，则不更新带宽值。我们通过下图来看一下过载检测器检测到的网络状态是如何影响状态机状态变化的。



我们可以看到，当过载检测器检测到过载时，状态机都切换到下降状态。这个很好理解，因为过载意味着实际带宽值小于发送码率了，需要调低带宽值，继而发送码率因为预估带宽值下降了也会下降。

当过载检测器检测到正常时，状态机都向上调一个状态。什么意思呢？

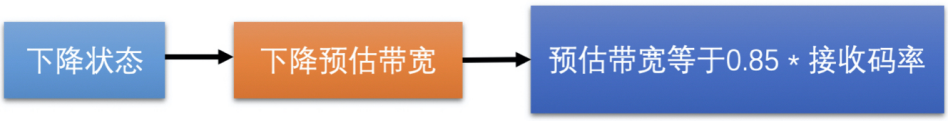
就是如果之前状态机处于下降状态，则更改为保持状态；如果状态机之前处于保持状态，则更改为上升状态；如果是上升状态那就不用变化了。这个也很好理解，因为过载检测器检测到正常，说明当前实际发送数据量还没有达到网络的最大承受量，则可以继续调高预估带宽值，或者之前在降低带宽值的话，就不再继续降低预估带宽值了。

当过载检测器处于欠载的状态时，状态机全部切换到保持状态。这个有点不好理解。欠载了不是需要直接提高预估带宽值的吗？为什么保持带宽不变呢？

这是因为 WebRTC 认为欠载主要发生前面因为过载了在网络设备中缓存了一定数据，之后网络状况变好了的时候，网络设备可以快速的发送完缓存中的数据，从而排空缓存。这个时候不要提高发送码率，等缓存排空之后，因为缓存带来的延时就会接近于 0 了，这对于降低端到端延时是很有用的，而过载检测器自然就会进入到正常状态。这样状态机就可以切换到上升状态，从而也就可以调高预估带宽值了。

好了，我们讲解了什么时候需要调高、什么时候需要降低带宽值。但是我们还没有讲到底调高多少以及降低多少。下面我们就来介绍一下具体调节预估带宽的公式。

带宽调整更新逻辑中，上调带宽逻辑比较复杂。因此，我们先介绍相对简单点的下降带宽调整。下降带宽其实很简单，就是预估带宽等于 0.85 倍的接收码率。



极客时间

但是下降带宽的时候需要做另外一个事情就是更新当前网络的最大带宽。因为处于下降带宽的过程中，说明当前发送数据量已经达到甚至超过了网络的承受能力。这个时候适合更新网络的最大带宽，将当前的接收码率与之前的最大带宽做加权平均求得当前的最大带宽，并更新最大带宽的标准差。这两个值之后调高带宽的时候需要用。

这个地方需要提一下，为什么预估带宽和最大带宽都是用接收码率来来计算的，那是因为发送码率可能会超出网络承受能力，不一定能很好的反映真实的网络带宽，相比而言，**接收端的接收码率更能够表示真实的网络带宽。**

在上升带宽的时候，是有两种情况的。

当前接收码率离最大带宽比较远，具体就是当前接收码率大于最大带宽加上 3 倍的最大带宽标准差，或者小于最大带宽减去 3 倍的标准差。这个时候应该快速调高预估带宽，我们使用的是乘性增的方式增加带宽值，就是每隔一秒钟将预估带宽乘以 1.08 倍。但是为了防止带宽大幅增长，预估带宽不能大于 1.5 倍的接收码率。

当前接收码率离最大带宽很近，具体就是当前接收码率大于最大带宽减去 3 倍标准差，并小于最大带宽加上 3 倍标准差。这时使用更小的幅度增加带宽，使用加性增的方式增加带宽值。预估带宽是每隔一个 RTT（往返时间，由其它模块计算得到，这里不展开讲）增加半个包的大小。如果一个包按照 1400 字节算的话，那就是每过一个 RTT 的时间就增加 700 字节。

好了，以上就是基于延时的带宽预测算法。

基于丢包的带宽预测算法

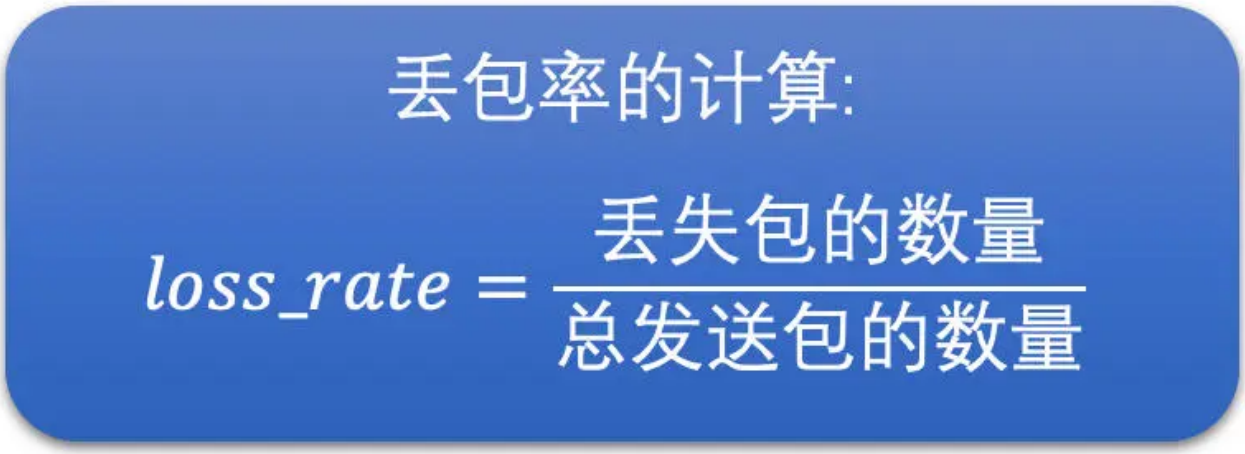
前面我们讲过，网络设备主要有两种：一种是有较大缓存的；一种是没有缓存或者缓存很小的。有较大缓存的网络设备在遇到数据量太大的时候会把数据放在缓存中，延迟发送。其表现就是网络延时加大。因此这种情况下我们基于延时做带宽预测是比较准确的。

而没有缓存或者缓存很小的网络设备，在遇到数据量太大的时候是直接将数据丢弃的，这会引起丢包率上升。为了能够更快更准确地做带宽预测，WebRTC 针对这种情况设计了基于丢包的带宽预测算法。

基于丢包的带宽预测算法相比基于延时的带宽预测算法简单很多，没有那么多步骤。**其整体思路就是根据 Transport-CC 报文反馈的信息计算丢包率，然后再根据丢包率的多少直接进行带宽调整更新。**

丢包率的计算

接收端会将接收到的每一个包的信息放到 Transport-CC 报文中，包括每一个 RTP 包的序号以及这个包有没有接收到。而没有接收到就代表这个包丢失了。这样就很简单了，发送端收到这个 Transport-CC 报文之后只需要统计这个 Transport-CC 报文总共有多少个包，以及丢失了多少个包，就可以计算得到丢包率了。



带宽调整

有了丢包率之后，我们就可以做带宽调整了。根据丢包率可以分三种情况。具体如下：

如果丢包率 < 2 %，认为当前网络状况很好，需要调高带宽值，带宽值等于过去 1 秒钟所有预测得到的带宽值的最小值 * 1.08；

如果 2% < 丢包率 < 10%，认为当前网络状况正常，不做带宽调整；

如果丢包率 > 10%，认为网络状况不好，需要降低带宽值，带宽值等于当前预估带宽值 * (1 - 0.5 * 丢包率)。

如果不太明白，没关系，下面的图能让你一目了然。

这就是基于丢包的带宽预测算法。

现在我们有了基于延时的带宽预测算法，主要用于有大缓存网络设备存在时的带宽预测；同时也有了基于丢包的带宽预测算法，主要用于有小缓存或无缓存网络设备存在时的带宽预测。由于互联网中一般是两种网络设备都有，因此，最终的预估带宽值等于这两者中的最小值。

到这里，我们的带宽预测算法最核心的部分就完成了。但是不知道你有没有发现一个问题，就是我们在网络变差的时候，预估带宽会快速的被下调，但是网络变好的时候预估带宽会比较缓慢的上升，同时如果当前发送码率比较小的话，预估带宽还会被限制，不能超过 1.5 倍的接收码率。

也就是说，如果我们当前视频处于静止画面的状态，发送的码率会很小。这样预估带宽就很难从一个比较小的带宽调整上去。如果此时画面突然动起来，即便实际网络带宽足够，还是会因为预估带宽不够而限制发送码率，从而导致画面出现模糊和马赛克等问题。

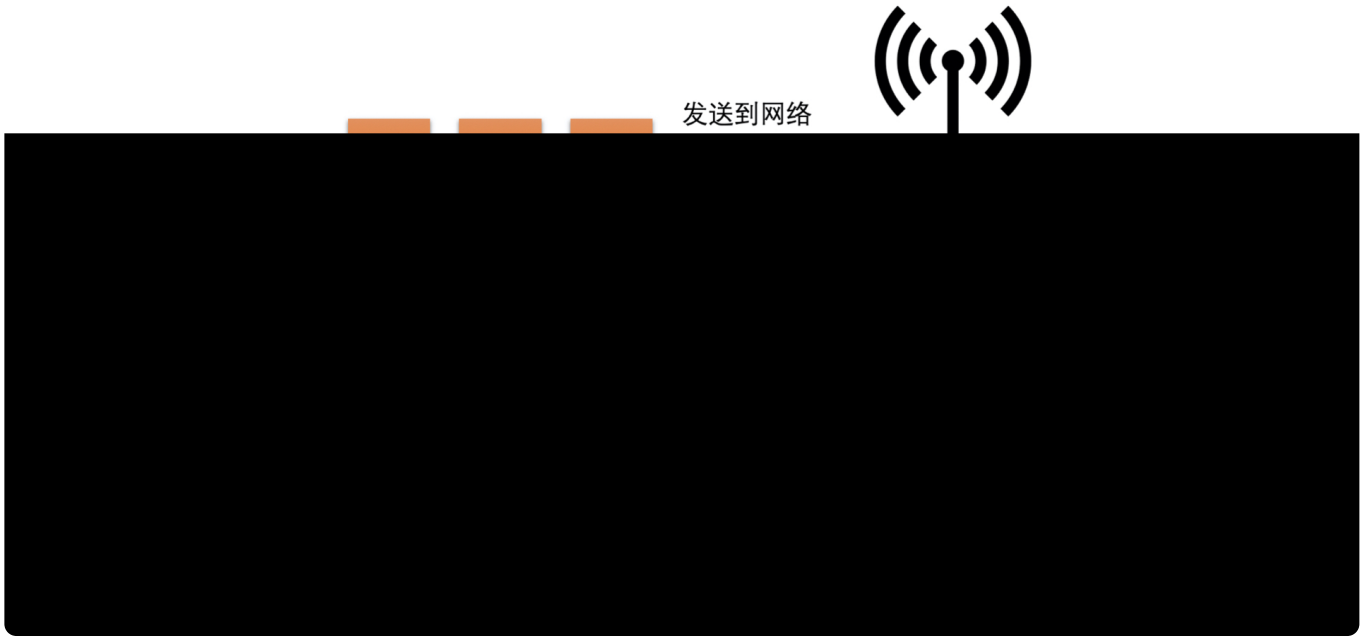
还有就是程序刚开始启动的时候，预估带宽的爬升也需要慢慢的加上去。但是在做音视频通信的时候，有可能一开始就会出现视频画面变化比较快的情况，这样可能刚开始的时候视频画面就会是模糊的或者有很多马赛克。

这种问题有没有方法解决呢？答案是有的，下面我们就来讲讲第三种带宽预测的算法，它可以帮助我们快速的探测出当前网络的最大带宽。

最大带宽探测算法

最大带宽探测算法相对来说思路也比较简单。它的主要过程是：

第一步，发送端设定一个探测的目标带宽，一般设置为当前带宽的 2 倍、3 倍或者 6 倍。发送端在发送数据的时候就以这个探测目标带宽的速度快速发送 RTP 包，一般发送时间（也叫做探测时间）是 15ms，同时将这段时间用于探测使用的 RTP 包信息保存在发送端（前面基于延时的带宽预测的时候，也需要保存发送的 RTP 包的发送时间等信息，其实都是保存在发送历史数据里面，只是探测的 RTP 包会多保存一些信息），并给这些 RTP 包标上是探测包的标记以及探测的 cluster_id，每一次探测使用的 cluster_id 都不同，用于区分每次探测的 RTP 包，防止多次探测时弄混了。



第二步，接收端并不关注当前包是不是探测包，而是直接统计每一个包的序号和接收时间，将统计结果组成 Transport-CC 报文反馈给发送端。

第三步，发送端接收到 Transport-CC 报文之后，会看报文中的每一个包是不是探测包，如果是探测包，就从发送端发送的历史数据中，取出其 cluster_id 和发送时间，并且从 Transport-CC 报文中得到接收时间，再把这些信息送入到探测带宽计算器中，当探测带宽计算器中相同 cluster_id 的 RTP 包信息数量达到一定值之后，就可以计算最终探测到的带宽值了。计算方法如下：

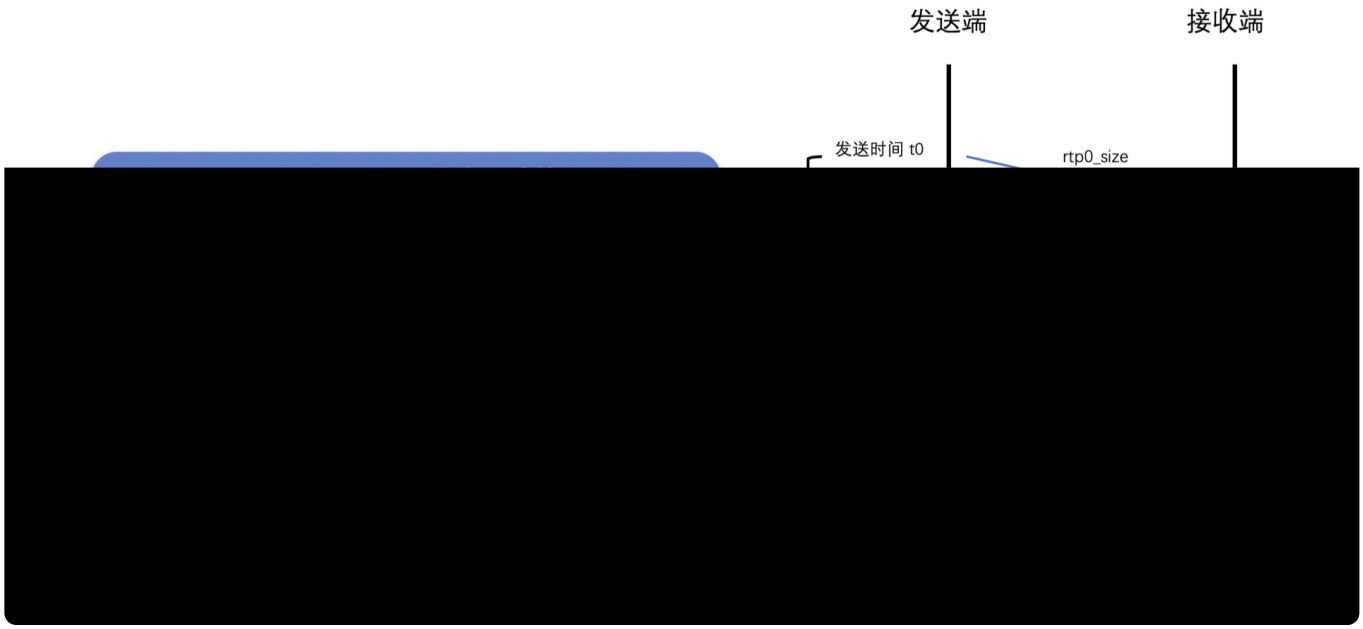
首先，根据发送的探测 RTP 包总大小，和探测 RTP 包的总发送时间，相除就得到了发送端的最大带宽值了。

然后，再根据接收端接收到的探测 RTP 包总大小，和探测 RTP 包的总接收时间，相除就得到了接收端的最大带宽值了。

之后，取两者中的最大值作为探测到的网络最大带宽值。同时，如果接收端的最大带宽小于 0.9 倍的发送带宽，说明当前探测目标带宽已经到了最大带宽了，为了保险一些，最后将最大带宽取为接收端最大带宽的 0.95 倍。

最后，如果当前过载检测器检测不是在过载状态的话，预估带宽值更新为探测到的最大带宽值。

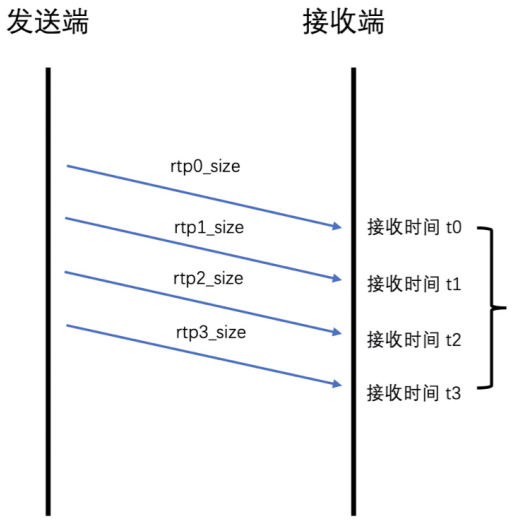
具体如图所示：



接收端最大接收带宽的计算:

$$probe_bw = \frac{rtp1_size + rtp2_size + rtp3_size}{t3 - t0}$$

注：rtp0的包大小没有被记入是因为它的接收持续时间也没有记入到计算当中

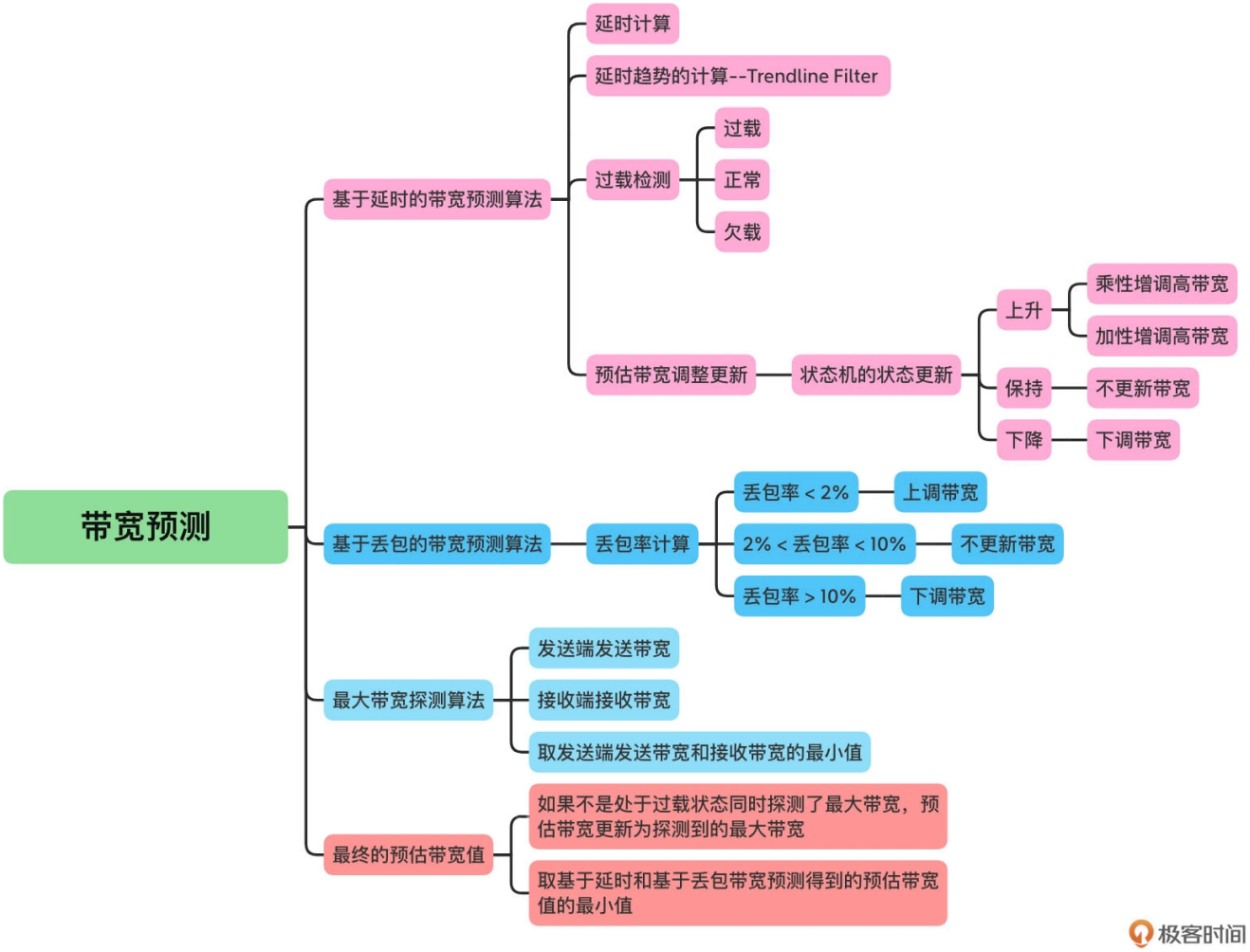


极客时间

这就是最大带宽探测算法，它一般是在程序刚开始启动的时候使用并在程序运行的过程中进行周期性的探测，每隔一段时间定时探测一下。其实如果你自己设计带宽预测算法的话，也可以在实际发送带宽很小的时候探测一下，防止出现因为实际发送码率很小，而无法准确预测网络当前最大带宽值的问题。

小结

为了帮助你记忆，这里我用图帮你总结了一下这节课的知识点。



WebRTC 中带宽预测主要分为基于延时的带宽预测算法、基于丢包的带宽预测算法以及最大带宽探测算法。

基于延时的带宽预测算法主要是解决网络中含有大缓冲网络设备场景的带宽预测。基于丢包的带宽预测算法主要是解决网络中有小缓冲或无缓冲网络设备场景的带宽预测。最终预估带宽等于这两者预测到的带宽值中的最小值。

同时，为了防止出现发送码率大幅低于实际网络带宽而导致网络带宽预估偏低的问题，我们还引入了最大带宽探测算法，可以周期性的探测网络的最大带宽。如果当前网络不是处于过载状态同时又探测到了最大带宽的话，就将预估带宽更新为探测到的最大带宽。

思考题

对于视频来说，带宽预测的最终目的是什么？

你可以把你的答案和感受写下来，分享到留言区，与我一起讨论。我们下节课再见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | RTP & RTCP：如何正确地将视频装进RTP中？

下一篇 11 | 码控算法：如何控制视频的编码码率？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。